Mustererkennung - Aufgabenblatt 04

André Hacker und Dimitri Schachmann

1. Prostate Cancer

Unser Programm geht folgendermaßen vor

1. Einlesen des Input (Trainingsdaten) und hinzufügen einer ersten Spalten mit Einsen
2. Berechnen der Koeffizienten (weights) gemäß Least Squares Fitting.
3. Berechnung der Fehlerrate

Folgende Fehler-Kennzahlen berechnet unser Programm. Das Fitting haben wir immer auf Basis der Trainingsdaten durchgeführt (Annahme, dass das gefordert war).

```
Fitting was always done based on Training data
Error-rate was determined for both, Training and Test data

Sum Squared Error (Training): 29.426
Sum Squared Error (Test): 15.638

Mean Squared Error (Training): 0.439
Mean Squared Error (Test): 0.521

Mean absolute Deviation (Training): 0.499
Mean absolute Deviation (Test): 0.523
```

Man sieht dass sich die Werte Mean-absolute-Deviation wesentlich besser für einen Vergleich eignen als die Summen der Quadrate. Außerdem sind sie in der gleichen Einheit wie der Output (lpsa)
Um zu verstehen, wie groß die Abweichung tatsächlich ist, haben wir die Output-Daten analysisert:

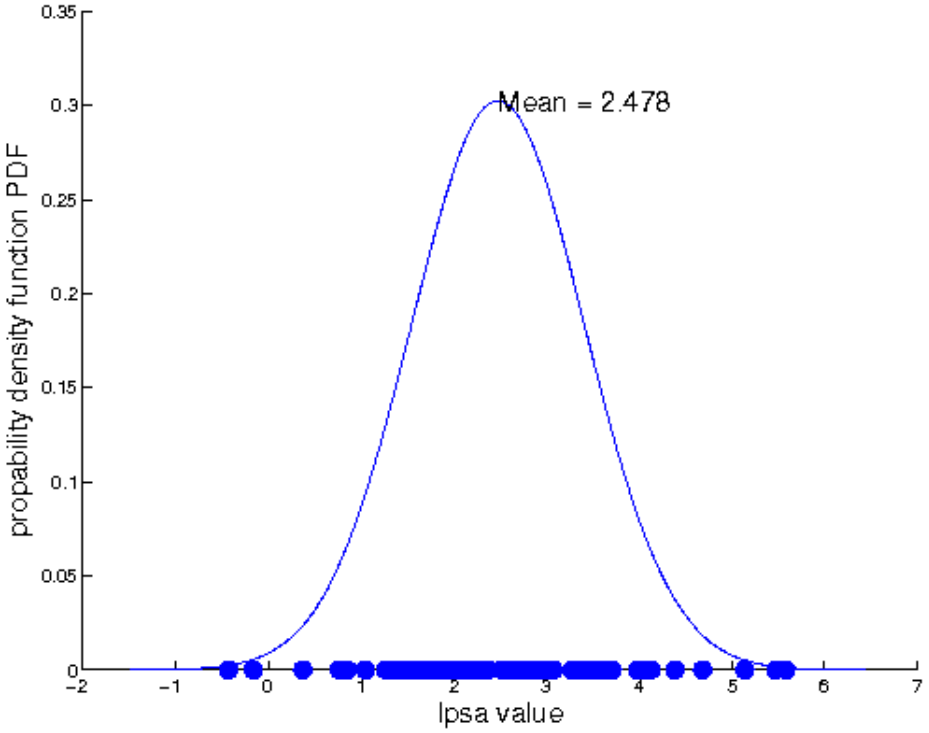


Abbildung 1: Verteilung des Outputs (Gauß-modeliert)

Source Code

```
% Parameter: lcavol lweight age lbph svi lcp gleason pgg45 lpsa train (F/T)

function linreg
  % =======
  % TASK 1
  % =======

  tra = dlmread('prostate-tra.mat', ' ');
  tes = dlmread('prostate-tes.mat', ' ');

  % Convenience
  traIn = tra(:,1:end-1);
  tesIn = tes(:,1:end-1);
  traOut = tra(:, end);
  tesOut = tes(:, end);

  % Plot distribution of lpsa
  plotOutputDistribution(traOut, tesOut);

  % Add 1 dimension for offset/bias
  traIn = [ones(size(tra, 1),1) traIn];
  tesIn = [ones(size(tes, 1),1) tesIn];
```

```
% 1) Compute weights minimizing the squared error
weights = getWeightsLeastSquares(traIn, traOut);

% 2) Compute squared error
sumErrorTra = sumSqError(weights, traIn, traOut);
sumErrorTes = sumSqError(weights, tesIn, tesOut);
% Nice to have:
meanErrorTra = meanSqError(weights, traIn, traOut);
meanErrorTes = meanSqError(weights, tesIn, tesOut);
mAbsDevTra = meanAbsDev(weights, traIn, traOut);
mAbsDevTes = meanAbsDev(weights, tesIn, tesOut);

%Write result to file
fid = fopen('task1-results.txt','w');
fprintf(fid, 'Fitting was always done based on Training data\n');
fprintf(fid, 'Error-rate was determined for both, Training and Test data\n\n');
fprintf(fid, 'Sum Squared Error (Training): %.3f\n', sumErrorTra);
fprintf(fid, 'Sum Squared Error (Test): %.3f\n\n', sumErrorTes);
fprintf(fid, 'Mean Squared Error (Training): %.3f\n', meanErrorTra);
fprintf(fid, 'Mean Squared Error (Test): %.3f\n\n', meanErrorTes);
fprintf(fid, 'Mean absolute Deviation (Training): %.3f\n', mAbsDevTra);
fprintf(fid, 'Mean absolute Deviation (Test): %.3f\n', mAbsDevTes);
fclose(fid);

% =======
% TASK 2
% =======
h1 = figure('Name','Sum Squared Errors for subsets (Training)','NumberTitle','off');
xlabel('Size of subset (k)', 'FontSize', 17);
ylabel('sum squared error', 'FontSize', 17);
hold on;
h2 = figure('Name','Mean squared Errors for subsets (Training)','NumberTitle','off');
xlabel('Size of subset (k)', 'FontSize', 17);
ylabel('mean squared error', 'FontSize', 17);
hold on;
h3 = figure('Name','Mean absolute Errors for subsets (Training)');
xlabel('Size of subset (k)', 'FontSize', 17);
ylabel('mean absolute error', 'FontSize', 17);
hold on;
h4 = figure('Name','Sum Squared Errors for subsets (Test)');
xlabel('Size of subset (k)', 'FontSize', 17);
ylabel('sum squared error', 'FontSize', 17);
hold on;

fid = fopen('task2-results.txt','w');
for k=1:8
  comb = nchoosek(1:8,k);
  results = zeros(size(comb,1), 1);
  listweights = zeros(size(comb,1), k+1);
  for l=1:size(comb,1)
    % Reduce to current parameters
    samplesTra = tra(:,comb(l,:));
```

```
      samplesTes = tes(:,comb(l,:));
      samplesTra = [ones(size(samplesTra, 1),1) samplesTra];
      samplesTes = [ones(size(samplesTes, 1),1) samplesTes];
      % apply lin. reg.
      weights = getWeightsLeastSquares(samplesTra, traOut);
      % compute error-measures
      sumErrorTes = sumSqError(weights, samplesTes, tesOut);
      sumErrorTra = sumSqError(weights, samplesTra, traOut);
      meanErrorTes = meanSqError(weights, samplesTes, tesOut);
      meanAbsErrorTes = meanAbsDev(weights, samplesTes, tesOut);
      figure(h1);
      scatter(k, sumErrorTra, 200, 'g+');
      figure(h2);
      scatter(k, meanErrorTes, 200, 'g+');
      figure(h3);
      scatter(k, meanAbsErrorTes, 200, 'g+');
      figure(h4);
      scatter(k, sumErrorTes, 200, 'g+');

      % Store combination and error rate in cell array to sort
      results(l, 1) = sumErrorTes;
      listweights(l, :) = weights;
    end
    printResults(results, comb, listweights, fid);
  end
  fclose(fid);

  print(h1,'-dpng','task2-sum-sq-errors.png');
  print(h2,'-dpng','task2-mean-sq-errors.png');
  print(h3,'-dpng','task2-mean-abs-errors.png');
  print(h4,'-dpng','task2-sum-sq-errors-tes.png');

  % =======
  % ADDON 1
  % Plot lin. regression for single features
  % =======
  for i=1:8
    plotRegressionForFeature(tra, i);
  end

end

% Least Squares Fitting based on samples and output column vector
function w = getWeightsLeastSquares(samples, y)
  % Compute pseudo-inverse matrix
  pseudo = getPseudoInverse(samples);
  w = pseudo * y;
end

% Get pseudo-inverse matrix, needed for least squares fitting
function p = getPseudoInverse(X)
  % very likely that inverse exists if we have many samples
  p = inv(X' * X) * X';
```

```
end

% Predicts value for input based on weights
% input = list of row vectors
% weights = column-vector
function r = predict(weights, samples)
  r = samples * weights;
end

% Return Sum of squared errors
function e = sumSqError(weights, samples, y)
  deviation = y - predict(weights, samples);
  e = deviation' * deviation;
end

% Get Mean Squared Error MSE
function e = meanSqError(weights, samples, y)
  e = sumSqError(weights, samples, y) / size(samples,1);
end

% Mean absolute deviations/errors (MAD, LAE)
% Though we did not use least absolute deviations method...
function e = meanAbsDev(weights, samples, y)
  e = sum(abs(y - predict(weights, samples))) ...
   / size(samples,1);
end

% Print results of subset task
function printResults(results, combs, weights, fid)
  % sort by error rate
  k = size(combs,2);
  [results, I] = sortrows(results);
  combs = combs(I,:);
  weights = weights(I,:);

  fprintf(fid, '\nk=%d\n', k);
  for i=1:size(results,1)
    fprintf(fid,'%s\t%.3f\t%s\n', ...
      mat2str(combs(i,:),0), ...
      results(i,1), ...
      mat2str(weights(i,:),3));
  end
end

% Plot correlation of one feature with output.
function plotRegressionForFeature(tra, feature)
  X = [ones(size(tra, 1),1) tra(:,feature)];
  weights = getWeightsLeastSquares(X, tra(:, end));
  Y = predict(weights, X);
  h10 = figure();
  hold on;
  xlabel('input x (value of feature)', 'FontSize', 17);
  ylabel('output (predicted vs real value)', 'FontSize', 17);
```

```
  scatter(X(:,2), tra(:,end), 'bo', 'filled', 'Displayname', 'Real lpsa');
  scatter(X(:,2), Y, 300, 'r*', 'Displayname', 'Predicted lpsa');
  title(['Feature ' mat2str(feature)], 'FontSize', 30);
  print(h10,'-dpng',['task2-feature' mat2str(feature) '.png']);
end

function plotOutputDistribution(traOut, tesOut)
  h = figure();
  hold on;
  xlabel('lpsa value', 'FontSize', 15);
  ylabel('propability density function PDF', 'FontSize', 15);

  % Just look at both, train and test:
  out = [traOut ; tesOut];

  %Plot values
  scatter(out, zeros(size(out,1),1), 100, 'bo', 'filled');

  %plot probability density function
  mu = getMean(out)
  sigma = getCovar(out, mu);
  ix = mu-3*sigma:0.01:mu+3*sigma;
  iy = gaussDensity(mu, sigma, ix)';
  plot(ix,iy);
  text(mu, gaussDensity(mu, sigma, mu) ,['Mean = ' mat2str(mu,4)], 'FontSize', 15);

  print(h,'-dpng',['output-distribution-tes.png']);
end

% compute prob. density for normal distribution
function p = gaussDensity(mu, sigma, data)
  normalize = 1 / (sqrt(2*pi) * sigma);
  p = zeros(size(data,1), 1);
  for i=1:size(data,1)
    p(i) = normalize * exp( -(data(i)-mu)^2 / (sigma^2) );
  end
end

% Computes mean based on matrix with observations
% Input: rows with training data for one class
% Output: Mean (row vector)
function m = getMean(data)
  m = 1/size(data,1) * sum(data);
end


% Compute Covariance matrix
function s = getCovar(samples, mu)
  % Normalize and compute covar
  samples = samples - (ones(size(samples,1),1) * mu);
  s = samples' * samples;
  s = s / size(samples,1);
```

```
  % make some noise;)
  s = s + (0.0001 * eye(size(s)));
end
```

2. Subset Selection

Im folgenden die Summe der quadratischen Abweichungen für alle Kombinationen. Gefittet wurde mit den Trainingsdaten, den Fehler haben wir mit Trainings und Testdaten ermittelt:

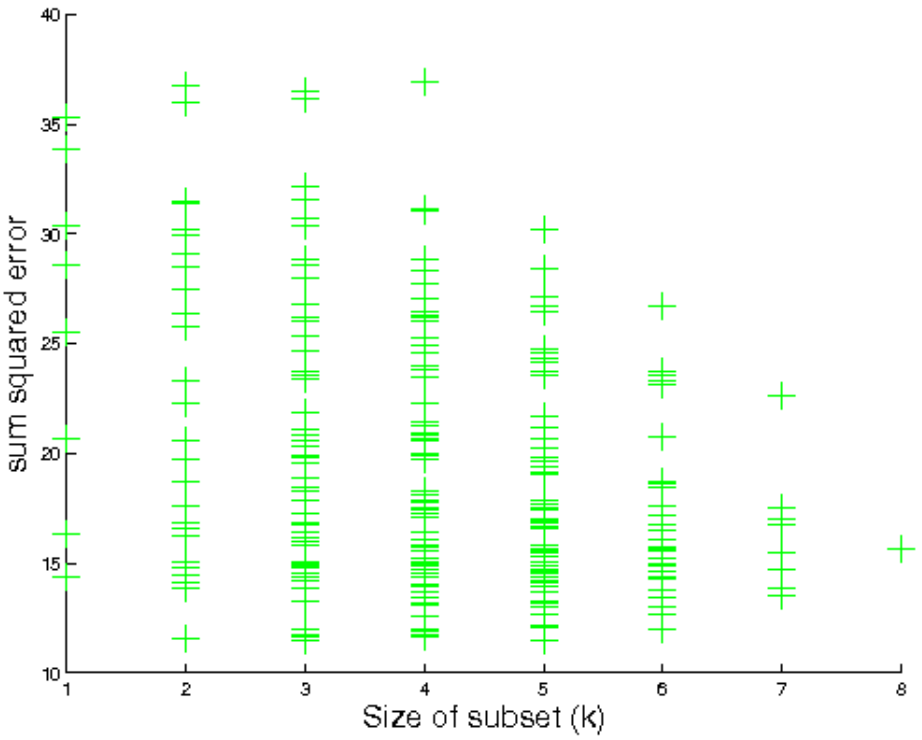


Abbildung 2: Summe der quadratischen Abweichungen auf Basis Testdaten

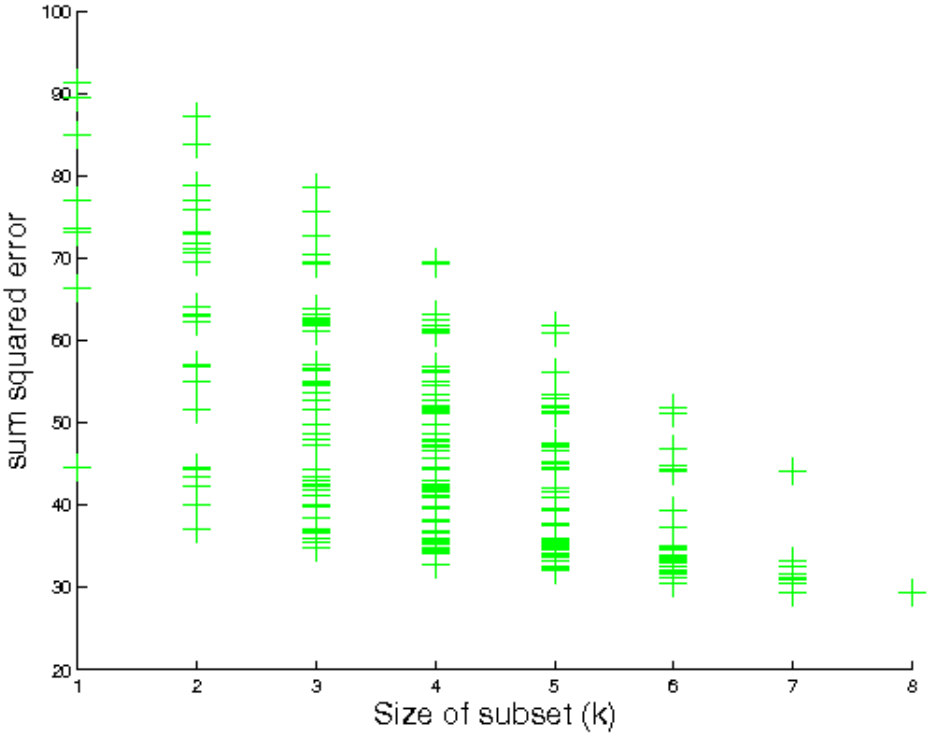


Abbildung 3: Summe der quadratischen Abweichungen auf Basis Trainingsdaten

Wir haben die Resultate für jedes Größe k des Subsets nach Fehlerrate sortiert (siehe Logfile unten) Die geringste Fehlerrate hinsichtlich der Testdaten wird mit den drei Features $\{1, 5, 7\}$ erzielt. Für die Fehlerrate auf den Trainingsdaten ist es optimal alle Parameter zu berücksichtigen. Es wird deutlich was mit dem Overfittings-Problem gemeint ist: Die Hyperebene die mit allen 8 Parametern für die Traningsdaten optimiert wurde hat zwar eine optimale Fehlerrate auf den Trainingsdaten, ist aber schlecht generalisierbar (für Testdaten).
Man sieht außerdem, dass z. B. das Feature 1 für alle k immer in den besten Subsets enthalten ist. Das deckt sich damit, dass Feature 1 scheinbar generell gut mit dem lpsa korreliert (siehe Feature-Plot für 1 ganz am Ende).
Die Reduktion der Merkmale sollte man aber wohl besser mit den in der Vorlesung vorgestellten Algorithmen (z. B. coefficient selection) durchführen.

Im Folgenden ein Auszug aus dem Logfile, aus dem die besten Subsets abgelesen werden können:

```
Features squared-error Weights
k=1
1 14.392 [1.52 0.713]
6 16.346 [2.54 0.422]
5 20.632 [2.09 1.6]
7 25.530 [-1.48 0.583]
8 28.617 [1.97 0.0185]
2 30.403 [-2.01 1.23]
3 33.847 [0.0795 0.0366]
4 35.299 [2.44 0.217]
```

```
k=2
[1 5] 11.584 [1.54 0.604 0.542]
[1 8] 13.828 [1.46 0.655 0.00504]
...

k=3
[1 5 7] 11.484 [1.27 0.595 0.536 0.0411]
[1 5 8] 11.632 [1.5 0.58 0.472 0.00332]
...

k=4
[1 3 5 7] 11.613 [1.12 0.591 0.00373 0.542 0.0291]
[1 2 3 5] 11.636 [-0.642 0.532 0.768 -0.00784 0.529]
...

k=5
[1 2 3 5 7] 11.498 [-1.48 0.503 0.812 -0.0124 0.502 0.151]
...

k=6
[1 2 3 5 7 8] 12.009 [-0.663 0.496 0.812 -0.0121 0.415 0.0114 0.00511]
...

k=7
[1 2 3 4 5 6 7] 13.493 [-0.866 0.556 0.619 -0.0187 0.144 0.803 -0.131 0.198]
...

k=8
[1 2 3 4 5 6 7 8] 15.638 [0.429 0.577 0.614 -0.019 0.145 0.737 -0.206 -0.0295 0.00947]
```

2. Anhang

Noch ein paar Ergänzungen (nicht Teil der Aufgabenstellung 2) Um ein Gefühl für die tatsächliche Abweichung zu bekommen haben wir die mittlere quadratische und absolute Abweichung berechnet:

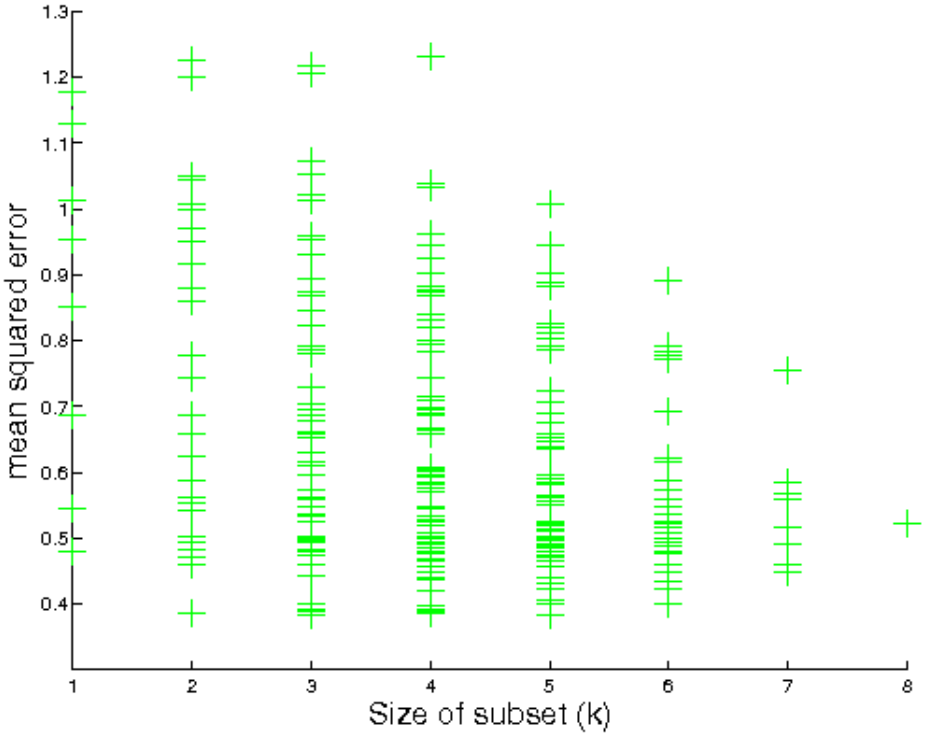


Abbildung 4: Mittelwert der quadratischen Abweichungen (Testdaten)

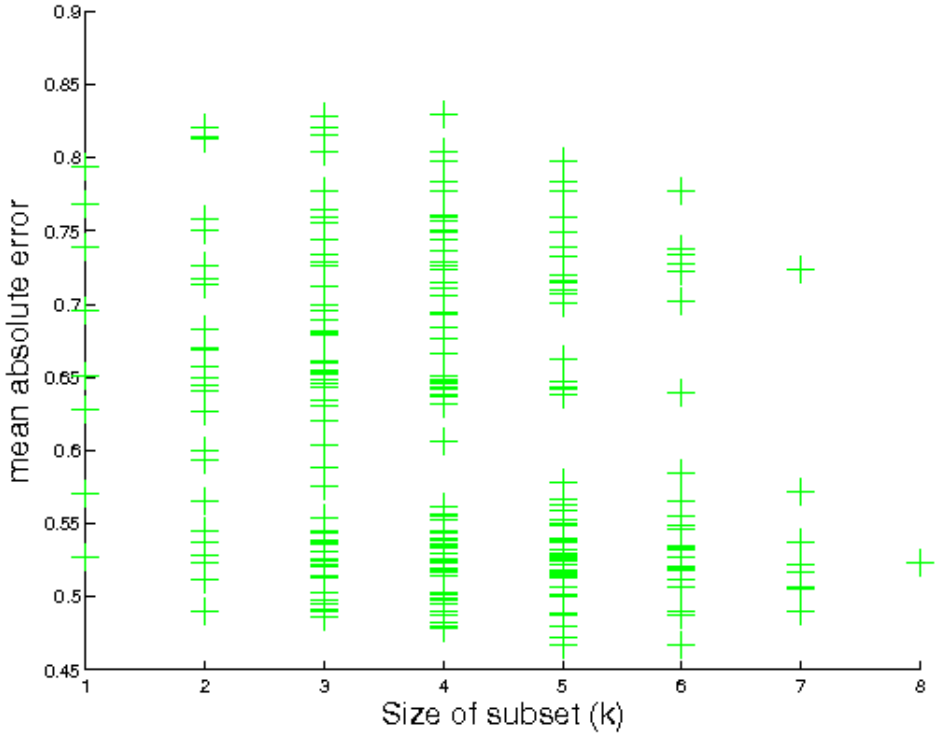
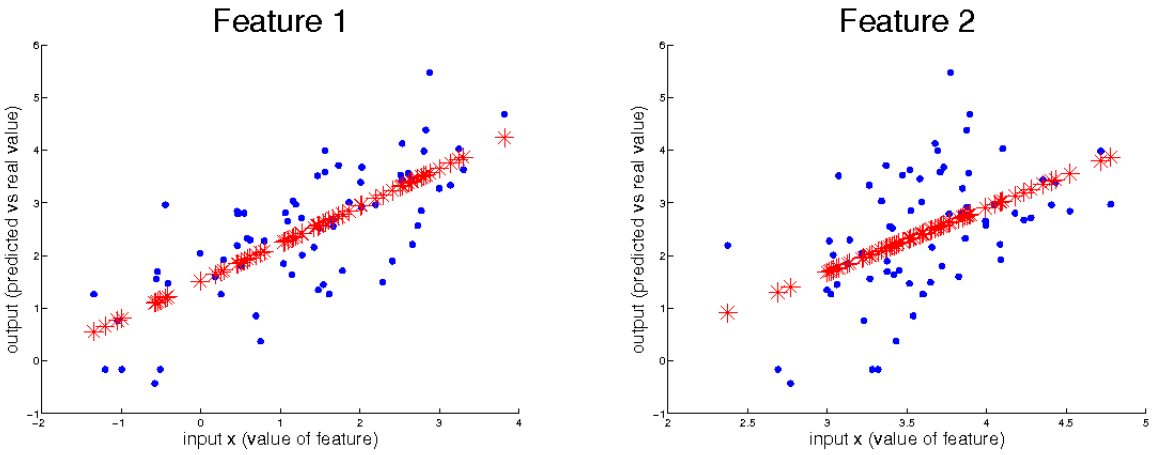
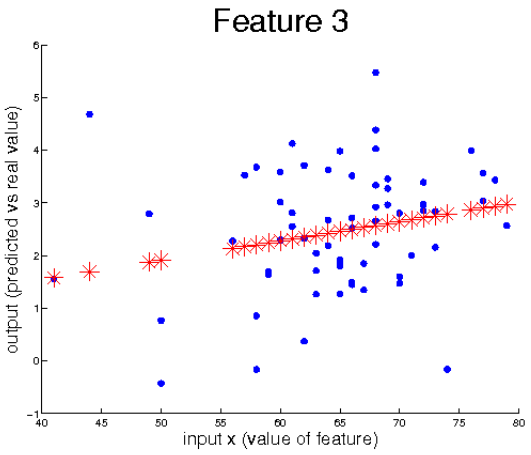


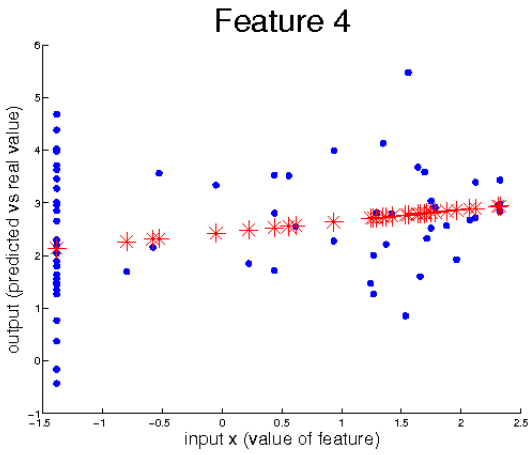
Abbildung 5: Mittelwert der absoluten Abweichungen (Testdaten)

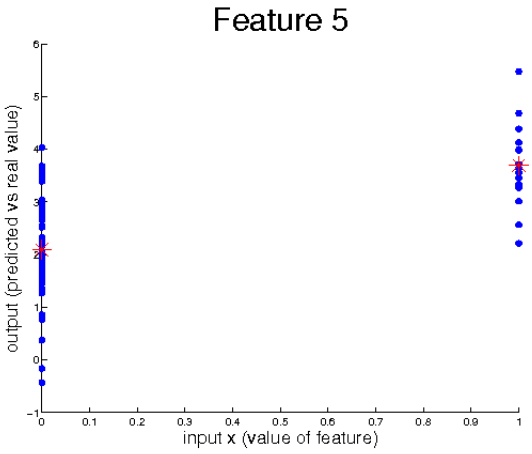
Erstaunlicherweise ist dass die absoluten Abweichungen den quadratische Abweichung beinahe vollständig entsprechen. Das liegt wohl daran, dass die Abweichungen mal kleiner, mal größer 0 sind und somit das quadrieren nicht zwangsweise in größeren Zahlen resultiert. Das es fast identisch ist, können wir uns aber nicht wirklich erklären.

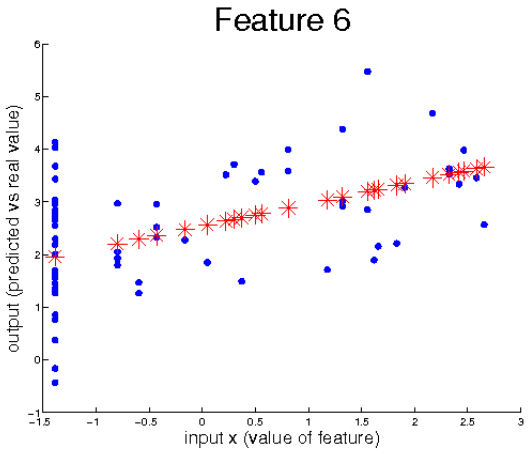
Außerdem haben wir die lineare Regression für jedes Feature einzeln geplottet:

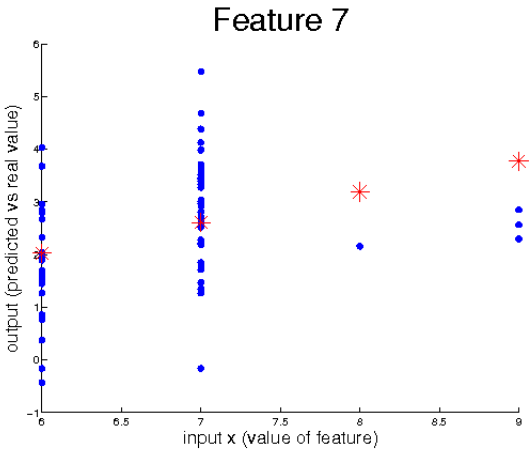


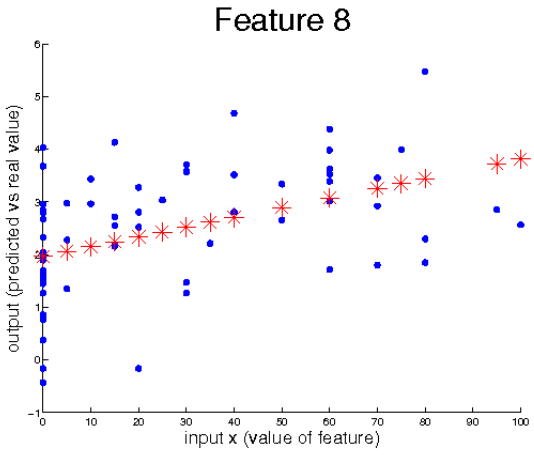

Feature 3
output (predicted vs real value)
input x (value of feature)


Feature 4
output (predicted vs real value)
input x (value of feature)


Feature 5
output (predicted vs real value)
input x (value of feature)


Feature 6
output (predicted vs real value)
input x (value of feature)


Feature 7
output (predicted vs real value)
input x (value of feature)


Feature 8
output (predicted vs real value)
input x (value of feature)